Lifted Backward Search for General Game Playing

Dave de Jonge and Dongmo Zhang

Western Sydney University
{d.dejonge, d.zhang}@westernsydney.edu.au

Abstract. A General Game player is a computer program that can play games of which the rules are only known at run-time. These rules are usually given as a logic program. General Game players commonly apply a tree search over the state space, which is time consuming. In this paper we therefore present a new method that allows a player to detect that a future state satisfies some beneficial properties, without having to explicitly generate that state in the search tree. This may lead to faster algorithms and hence to better performance. Our method employs a search algorithm that searches backwards through formula space rather than state space.

1 Introduction

The development of programs that can play specific games such as Chess and Go, has long been an important field in AI research. However, such game players are limited in the sense that they are only able to play one specific game and often rely on game-specific heuristics invented by humans. Therefore, recently more attention has been given to the concept of *General* Game Playing (GGP). A General Game Playing agent is able to interpret the rules of a game at run-time and devise a strategy for it without any human intervention. Since 2005 General Game Playing competitions have been held annually at the AAAI Conference [6]. A language called Game Description Language (GDL) [11] was invented to write down game-rules in a standardized, machine-readable way. GDL specifies games as a logic program, and is similar to other logic-based languages such as ASP [4] and Datalog [3].

Most GGP players apply a generic search algorithm such as minimax search [12], alpha-beta pruning [9] and, most importantly, Monte Carlo Tree Search (MCTS) [10, 7]. These techniques are based on the principle of forward searching: they start with the initial state, determine the set of legal moves in that state, then determine for each of these moves the next state that would result if that move were executed, evaluate this hypothetical future state, and then repeat the procedure.

In order to be able to quickly generate a future state given the current state and a legal move, most GGP players translate a GDL-specified game into a state transition machine. A disadvantage of this technique is that it requires *grounding* to get rid of any variables in the rules. Every rule is replaced by several variable-free copies of that rule by replacing every variable with a possible ground term for that variable. Grounding makes logical reasoning much simpler, but it also causes the size of the search space to explode, and may cause explicit symmetries in the game rules to get lost.

In this paper we propose a technique that may improve the speed of tree-search algorithms. The idea is that one may be able to evaluate a game state before it is generated

in the tree. We achieve this by applying a backward search that does not require grounding. It starts with a formula that describes states satisfying some desired property (such as being a terminal winning state) and then searches for formulas that describe states which are one move away from a state with that property, and so on. This backward search is applied before the forward search is started.

In the field of planning such a technique is known as Lifted-Backward-Search [8]. The difference between planning and GGP, however, is that planning is normally concerned with a single agent, or a group of agents with a common goal. In GGP on the other hand the agent needs to deal with adversaries. Therefore, our approach combines Lifted-Backward-Search with a min-max strategy. We show that our approach is sound and compete, in the sense that, given any desired property every state satisfying that property can be found by our algorithm, and that every state found by our algorithm indeed satisfies that property.

The rest of the paper is organized as follows: In Section 2 we give a short overview of existing work. In Section 3 we explain how our technique can be used in a GGP agent. In Section 4 we will formalize the language in which we present our algorithm. In Section 5 we will present the algorithm itself and prove its main properties. Finally, in section 6 we present our conclusions.

2 Related Work

FluxPlayer [13], the winner of the 2006 AAAI GGP competition is a player that applies an iterated deepening depth-first search method with alpha-beta pruning, and uses Fuzzy logic to determine how close a given state is to the goal state. Cadia Player [5], the winner in 2007, 2008, and 2012, is based on MCTS, extended with several heuristics to guide the rollouts so that they are better informed and hence give more realistic results, and also the winner of the 2014 competition, Sancho,[1] as well as the winner of 2015, Galvanise,[2] apply variants of MCTS.

A technique similar to ours is described in [14]. The main difference however is that their backward search is grounded. Furthermore, their algorithm only goes 1 step back, whereas our algorithm may take any number of backward steps.

In [2] the authors implement a heuristic backward search algorithm called HSPr for single-agent planning domains. They do not find much benefit in backward search however, because it generates many 'spurious states': states that are impossible to reach. However, in [1] the authors manage to improve HSPr by extrapolating several common techniques from forward search to backward search and thus creating a new regression planner called FDr. A general overview of planning algorithms, including Lifted Backward Search, can be found in [8]. To the best of our knowledge lifted backward search has never been used in domains with adversarial agents.

3 A Player Based on Backward Search

A standard approach to General Game Playing is to apply Monte Carlo Tree Search. Key to this approach is that one evaluates a game state w by simulating a game in which

[1] `http://sanchoggp.blogspot.co.uk/2014/05/what-is-sancho.html`
[2] `https://bitbucket.org/rxe/galvanise_v2`

random moves are played starting at state w until a terminal state is reached, and then evaluating the player's utility for that terminal state. This is called a *rollout*. Repeating this many times and averaging the utility values returned by these rollouts yields an estimation of the true utility value of the state w. Unfortunately, since these rollouts are random, one needs to perform a lot of them before the result becomes accurate.

We propose to increase the accuracy by combining MCTS with Lifted-Backward-Search. The idea is that, before applying MCTS, the player will apply backward search for a given amount of time, say 10 seconds for example. Assuming the game is a turn-taking game for two players, which we refer to as A and B, the backward search generates two sequences of formulas $\alpha_0, \alpha_1, \alpha_2 \ldots$ and $\beta_0, \beta_1, \beta_2 \ldots$ which are stored in memory and will be used by the rollouts of the MCTS. Here, α_0 represents the set of winning states for our player A, while α_1 represents the set of all states for which player A has a move that will lead to winning state. The formula α_2 represents the set of states for which the opponent cannot avoid the next state to satisfy α_1. That is, in general, if a state satisfies α_i it means that A has a strategy that can guarantee that the next state will satisfy α_{i-1} (either because A has a move that leads to such a state or because B does not have any move that can lead to a state that does not satisfy it). The formulas β_i have the opposite interpretation: β_0 represents a winning state for the opponent B, and if a state satisfies β_i our opponent can enforce the next state to satisfy β_{i-1}. After the backward search our player applies MCTS. For each game state explored during a rollout, the algorithm determines whether it satisfies any of the α_i or β_i. If it does satisfy any of these formulas the rollout can be stopped and return either the value 100 (if the state satisfies any α_i), or 0 (if it satisfies some β_i).

Not only does this allow us to terminate the rollout earlier, it also yields a much more accurate result. Say for example that we have a state w in which player A has 20 possible moves, of which only one leads to a winning state yielding 100 points, and all others lead to a draw, yielding 50 points. Of course, any rational player would always pick that winning move, so the non-terminal state w itself can be assigned a value of 100. However, since rollouts are random, a rollout will only pick the correct winning move and return 100 once in every 20 times it visits w. With our algorithm on the other hand, the rollout will detect that w satisfies α_1 and therefore always return the correct utility value of 100.

Our backward search works with non-grounded formulas, which means that many different states can be described by a single formula. Of course, generating those formulas may take a lot of time, and in the worst case the size of the formulas α_n may grow exponentially with n. However we expect that in many cases the formulas remain relatively compact because they contain variables.

The fact that α_0 and β_0 represent terminal winning states for the respective players is just an example. One could also give them the interpretation of some other important game property. For example, in chess, α_0 could represent those states in which A just captured the opponent's queen. In that case, whenever the rollout function finds a state that satisfies any of the α_i's it will return some heuristic value.

4 Formal Definitions

In general, GGP deals with games that take place over one or more discrete rounds. In each round the game is in a certain state, and each player chooses an action (also referred to as a *move*) to take. The game description specifies the initial state, which actions each player is allowed to choose given any state, and how in any state the chosen actions determine the next state.

In this paper we assume the game is a two-player turn-taking game that can end in three possible ways: a win for the first player, a win for the second player, or a draw. Furthermore, we assume that if the game has a winner, then the winner is always the player who made the last move. Examples of such games are Chess, Checkers and Tic-Tac-Toe. In fact, it seems that most of the games available on the GGP website[3] satisfy these criteria. Furthermore, we also assume that the game description does not contain any cycles (see Def. 4). This is a restriction, because GDL only requires rules to be *stratified* (see [11]), which is a weaker assumption than being cycle-free. We do expect however that the assumption of being cycle-free can be dropped, but we leave it as future work to investigate this.

We will refer to the two players by A and B, where A is the player that applies our algorithm and B is our opponent. Since we assume turn-taking games in each round there is only one player that has an action to choose. We call this player the *active* player of that round.

4.1 State Transition Model

We first define the notion of a *game* in terms of a state transition model.

Definition 1. *A **game** $\mathcal{G}$ is a tuple $\langle P, \mathcal{A}, W, w_1, T, L, u, U \rangle$, where:*

- *P is the set of* players: $P = \{A, B\}$
- *$\mathcal{A}$ is the set of* actions.
- *W is a non-empty set of* states.
- *$w_1 \in W$ is the* initial *state.*
- *$T \subset W$ is the set of* terminal *states.*
- *$L : W \setminus T \to 2^{\mathcal{A}}$ is the* legality *function, which describes for each non-terminal state which actions are legal.*
- *$u : W \times \mathcal{A} \to W$ is the update function that maps each state and action to a new state.*
- *$U : T \times P \to [0, 100]$ is the utility function that assigns a utility value to each terminal state and player.*

The idea is that the players A and B both have their own set of actions, denoted $\mathcal{A}_A$ and $\mathcal{A}_B$ respectively, with $\mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_B$ and $\mathcal{A}_A \cap \mathcal{A}_B = \emptyset$. The game consists of discrete rounds, in which the players alternately pick an action a from their respective action sets and hence generate a sequence of actions $a_1, a_2, \ldots a_n$. Given a state w and an action a the update function u defines a new state $u(w, a)$. Therefore, given the initial state w_1 and a sequence of actions the update function defines a sequence of states $w_1, w_2, \ldots$

[3] `http://www.ggp.org/view/tiltyard/games/`

where each w_{i+1} equals $u(w_i, a_i)$. If the current state is w_i then the players may only pick their action from the set $L(w_i)$. Since we are assuming turn-taking games it is always the case that either $L(w) \subseteq \mathcal{A}_A$ or $L(w) \subseteq \mathcal{A}_B$.

4.2 Syntax

In this paper we will use our own language $\mathcal{L}$, which is a segment of first-order logic, because it is easier to describe our algorithm in $\mathcal{L}$ than in GDL. Our language borrows its basic components from GDL so that any game description given in GDL can be translated easily into $\mathcal{L}$. Given a game $\mathcal{G}$ we define $\mathcal{L}$ to be a language with a finite set of constants, function symbols and relation symbols which are specific to the game $\mathcal{G}$, a finite set of variables and logical connectives ($\neg$, $\vee$, $\wedge$, $\rightarrow$ and $\exists$).[4] $\mathcal{L}$ inherits the following game-independent relation symbols from GDL: $distinct$, $true$, $does$, $legal$, $goal$, $terminal$, and $next$. Their semantics will be given in the next subsection.[5] As a segment of first-order logic, we do not employ the full syntax of first-order logic. Instead we impose heavy restrictions on the structure of terms and formulas. A *formula* in $\mathcal{L}$ can either be a *complex formula*,[6] which is a combination of atoms using $\neg$, $\vee$, $\wedge$, or $\exists$, or a *rule* defined as follows:

Definition 2. *A **rule** is an expression in $\mathcal{L}$ of the form: $p_1 \wedge p_2 \wedge \ldots p_n \rightarrow q$ where each p_i is a positive or negative literal, and q is a positive literal. The atom q is called the **head** of the rule and the p_i's are called the **subgoals** of the rule. The conjunction of subgoals is called the **body** of the rule.*

The body of a rule may be the empty conjunction (in which case the rule is also called a *fact*), and the subgoals and the head of a rule may contain variables. For clarity we will always denote variables with a question mark, e.g. $?x$. Similar to the restrictions on rules in GDL, the relation symbols $distinct$, $true$ and $does$ cannot appear in the head of any rule, while $legal$, $goal$, $terminal$ and $next$ cannot appear in the body of any rule. Apart from these key words rules may also contain user-defined relation symbols.

Definition 3. *There is a certain subset of the constants of $\mathcal{L}$ that we call **action-constants**, and a certain subset of the function symbols that we call **action-functions**. An action-function can only contain action-constants or variables that range over the action-constants. A term containing an action-function is called an **action-term**.*

The relation symbols $does$ and $legal$ can only contain action terms. A ground atom of the form $does(t)$ is called an *action-proposition*. The size of the set of ground action terms must be equal to the size of $\mathcal{A}$.

Definition 4. *The **dependency graph** of a set of rules $\mathcal{R}$ is a directed graph that contains a vertex v_p for each relation symbol p that appears in any rule in $\mathcal{R}$, and there is an edge from v_p to v_q if there is a rule in which p appears in the body and q appears in the head. A set of rules $\mathcal{R}$ is **cycle-free** if its dependency graph does not contain any cycles.*

[4] The universal quantifier $\forall$ is not included in the language.

[5] GDL defines more relations, but these are not relevant for this paper.

[6] The implication $\rightarrow$ is not allowed in a complex formula.

Definition 5. *A proposition of the form $true(t)$, where t is a ground term of $\mathcal{L}$, is called a **base proposition**. The (finite) set of all base propositions is denoted as $\mathcal{B}$.*

Definition 6. *A formula is called **unwound** if it only contains the relation symbols 'distinct', 'true' and 'does', and it is called **wound** otherwise.*

We will see below that for an unwound formula we can determine whether it is satisfied or not in a straightforward manner, whereas for wound formulas we first need to rewrite ('unwind') the formula as an unwound formula, using the rules of the game.

Definition 7. *Let φ be an atom and p be the relation symbol of φ. Then φ is called **statewise** if none of the paths in the dependency graph of $\mathcal{R}$ that go through v_p pass through the vertex corresponding to the relation symbol 'does'. A formula ϕ is called statewise if all its atoms are statewise. A formula is called **non-statewise** otherwise.*

As we will see below, the satisfaction of a statewise formula only depends on the game state, while for non-statewise formulas it depends on the state as well as on the chosen action. Just like in GDL, we pose the restriction on $\mathcal{L}$ that any atom containing *legal*, *goal*, or *terminal* must be statewise.

4.3 Semantics

Given a game $\mathcal{G}$ and a language $\mathcal{L}$ for that game, let V be a valuation function. That is: V is an injective function $V : W \to 2^{\mathcal{B}}$ that maps each world state of $\mathcal{G}$ to a different set of base-propositions. The interpretation of V is that when the game is in a state w the propositions in $V(w)$ are considered true and all other base-propositions are considered false. Furthermore, we define a bijective map μ that maps each action $a \in \mathcal{A}$ to a ground action-term t_a. In the following we will assume that $\mathcal{G}$, $\mathcal{L}$, $\mathcal{R}$, V, and μ are fixed.

Definition 8. *Let t and s be ground terms, $w \in W$, and $a, b \in \mathcal{A}$, then we define:*

- $V, \mathcal{R} \vDash_{(w,a)} true(t)$ *iff* $true(t) \in V(w)$
- $V, \mathcal{R} \vDash_{(w,a)} does(\mu(b))$ *iff* $a = b$ *and* $a \in L(w)$
- $V, \mathcal{R} \vDash_{(w,a)} distinct(t, s)$ *iff* $t \neq s$

Here $a = b$ means that a and b are syntactically equal, and $t \neq s$ that t and s are syntactically different. Note that the entailment of $true(t)$ and $distinct(t, s)$ does not depend on the action a.

Definition 9. *Let ϕ be any (non-ground) formula. Then we define $V, \mathcal{R} \vDash_{(w,a)} \exists\phi$ to hold iff there exists a substitution θ such that $\phi[\theta]$ is ground and $V, \mathcal{R} \vDash_{(w,a)} \phi[\theta]$ holds.*

Definition 10. *Let $\mathcal{R}$ be a set of rules, q an atom, r a rule in $\mathcal{R}$ and θ the most general substitution that unifies q with the head of r. Then we say the body of $r[\theta]$ is a **premise** of q. Furthermore, we say the disjunction of all premises of q is the **complete premise** of q, and is denoted as $Pr(q)$.*

For example, if we have an atom $q(t)$ where t is a ground term, and we have the following two rules: $p_1(?x) \to q(?x)$ and $p_2(?y) \to q(?y)$ then $Pr(q(t)) = p_1(t) \vee p_2(t)$. Note that this definition implies that if there is no rule r such that q can be unified with the head of r then $Pr(q) = \bot$, and that if q can be unified with a fact then $Pr(q) = \top$.

Definition 11. *Let q be any ground wound atom. Then we define:*
$V, \mathcal{R} \vDash_{(w,a)} q$ *iff* $V, \mathcal{R} \vDash_{(w,a)} \exists Pr(q)$.

Definition 12. *Let ϕ be any ground formula. Then $V, \mathcal{R} \vDash_{(w,a)} \phi$ is defined by the standard interpretation of the connectives of propositional logic, and Defs. 8 and 11.*

Definitions 8, 9, 11 and 12 together recursively define satisfaction. The fact that $\mathcal{R}$ is finite and cycle-free guarantees that the recursion terminates. If for some formula ϕ we have $V, \mathcal{R} \vDash_{(w,a)} \exists\phi$ then we say that (w, a) satisfies ϕ. If ϕ is statewise we may also say that w satisfies ϕ, and we may denote this as $V, \mathcal{R} \vDash_w \exists\phi$.

For example, if $\phi = goal(A, 100)$, and $\mathcal{R}$ contains a rule $true(t) \rightarrow goal(A, 100)$. Then w satisfies ϕ if $true(t) \in V(w)$.

Definition 13. *A game description G for a game $\mathcal{G}$ is a tuple $\langle \mathcal{L}, V, \mu, \mathcal{R} \rangle$ where $\mathcal{R}$ is finite and cycle-free. Furthermore, all of the following must hold:*

- $V, \mathcal{R} \vDash_{(w,a)} terminal$ *iff* $w \in T$
- $V, \mathcal{R} \vDash_{(w,a)} legal(\mu(a))$ *iff* $a \in L(w)$
- $V, \mathcal{R} \vDash_{(w,a)} next(t)$ *iff* $V \vDash_{u(w,a)} true(t)$
- $V, \mathcal{R} \vDash_{(w,a)} goal(p, x)$ *iff* $U(w, p) = x$

5 Backward Search

The goal of player A is to bring about a state in which both $terminal$ and $goal(A, 100)$ are satisfied. For a given state it is easy to verify whether these are satisfied or not. However, we would like our player to determine in advance whether it can enforce these propositions to be true in any future state. We therefore apply an algorithm that determines a sequence of unwound statewise formulas α_0, α_1, $\alpha_2 \ldots$ until time runs out. Their interpretation is that if a state w satisfies α_i then A can enforce a victory in i rounds. Note that our assumption that the winner always makes the last move implies that if the game is in a state that satisfies α_i and i is an odd number, then A is the active player, whereas if i is an even number then B is the active player. In order to explain how these formulas are calculated we need to define two operators, which we call the C-operator and the N-operator.

5.1 The C-Operator

In our language any formula can be rewritten as an equivalent unwound formula. Therefore, we here define an operator that takes a formula ϕ as input and outputs an equivalent unwound formula. In Section 5.2 we will see that this is important because the N-operator can only operate on unwound formulas.

Definition 14. *We define the C-operator as follows:*

- *For any terms t, s:* $C(true(t)) = true(t)$, $C(distinct(t, s)) = distinct(t, s)$.
- *For any action-term a:* $C(does(a)) = does(a) \wedge legal(a)$
- *For any wound atom q:* $C(q) = \exists Pr(q)$
- *For any non-atomic ϕ we obtain $C(\phi)$ by replacing each atom p in ϕ by $C(p)$.*

Thus, if we have: $\phi = q(t) \wedge does(a)$ with t a ground term, and the only rules in $\mathcal{R}$ of which the head can be unified with $q(t)$ are the following two: $p_1(?x) \rightarrow q(?x)$ and $p_2(?y) \rightarrow q(?y)$, with $?x$ and $?y$ variables, then:
$C(\phi) = (p_1(t) \vee p_2(t)) \wedge does(a) \wedge legal(a)$.

We use the notation $C^2(\phi)$ to denote $C(C(\phi))$, and $C^n(\phi)$ for $C(C^{n-1}(\phi))$. Since we require that $\mathcal{R}$ is finite and cycle-free, we have that for any formula ϕ there is always some n for which $C^n(\phi) = C^{n-1}(\phi)$. In other words, the sequence $C^1(\phi), C^2(\phi) \ldots$ always converges. Therefore, we can define $C^\infty(\phi)$ to be the limit of this sequence.

Definition 15. *We define C^∞ as follows: $C^\infty(\phi) = C^n(\phi)$ iff $C^n(\phi) = C^{n-1}(\phi)$*

Lemma 1. *For any formula ϕ the formula $C^\infty(\phi)$ is unwound.*

Proof. Suppose that the formula $C^n(\phi)$ contains an atom p of which the relation symbol is not $distinct$, $true$, or $does$. Then $C^{n+1}(\phi)$ contains $\exists Pr(p)$ instead of p, but $\exists Pr(p)$ is not equal to p because $\mathcal{R}$ is cycle-free. This means $C^n(\phi)$ is not equal to $C^{n+1}(\phi)$, which means that $C^n(\phi)$ is not $C^\infty(\phi)$. The conclusion is that if $C^n(\phi) = C^\infty(\phi)$ then $C^\infty(\phi)$ cannot contain any such atom, and thus is unwound.

Lemma 2. *For any formula ϕ, any state w and any action a we have:*
$V, \mathcal{R} \vDash_{(w,a)} \exists C^\infty(\phi)$ *iff* $V, \mathcal{R} \vDash_{(w,a)} \exists \phi$

Proof. It follows directly from Defs. 11, 9, 12 and 14, that $V, \mathcal{R} \vDash_{(w,a)} \exists C(\phi)$ iff $V, \mathcal{R} \vDash_{(w,a)} \exists \phi$. This argument can be repeated to prove the lemma.

Let η_A denote some wound statewise formula that player A desires to be satisfied, for example: $\eta_A = terminal \wedge goal(100, A)$ (this example formula describes the terminal states in which A wins the game). Then we define: $\alpha_0 := C^\infty(\eta_A)$. Note that α_0 describes exactly the same property as η_A, but α_0 is unwound.

5.2 The N-Operator

Now that we have defined α_0 we want to define the other formulas α_i. For this we need an operator that 'translates' a formula ϕ into a new formula ϕ' such that we know that if ϕ' is true in the current state then ϕ will be true in the next state.

Definition 16. *Given any unwound statewise formula ϕ, the formula $N(\phi)$ is obtained by replacing every occurrence of the relation symbol 'true' by the relation symbol 'next'. The resulting formula $N(\phi)$ is wound and non-statewise.*

For example, if $\phi = true(t_1) \vee true(t_2)$ then $N(\phi) = next(t_1) \vee next(t_2)$.

Suppose we have a game state w and we want to pick an action a such that the next state $u(w, a)$ will satisfy some formula ϕ. The action we need to pick is then the action a for which (w, a) satisfies $N(\phi)$. Specifically, if we apply this to α_0 it means that if a state-action pair (w, a) satisfies $N(\alpha_0)$ then A can guarantee the property α_0 to be satisfied in the next state by playing action a in state w.

Lemma 3. *Let ϕ be an unwound statewise formula. Then we have:*
$V, \mathcal{R} \vDash_{(w,a)} \exists N(\phi)$ *iff* $V, \mathcal{R} \vDash_{u(w,a)} \exists \phi$

Proof. We will only prove this for a specific example, but it can be generalized easily. Suppose that $\phi = true(t_1) \wedge true(t_2)$. We have that (w, a) satisfies $N(\phi) = next(t_1) \wedge next(t_2)$ iff (w, a) satisfies $next(t_1)$ and (w, a) satisfies $next(t_2)$. But, according to Def. 13 this is true iff $u(w, a)$ satisfies $true(t_1)$ and $u(w, a)$ satisfies $true(t_2)$, which means that $u(w, a)$ satisfies $true(t_1) \wedge true(t_2)$, which is ϕ.

For example, in Tic-Tac-Toe, suppose we have: $\phi = true(cell(1, 1, X))$ which states that ϕ is satisfied if the top left cell contains an X. Then we have:

$$N(\phi) = next(cell(1, 1, X)).$$

By itself this formula is not very useful, but by using the C^∞-operator we can transform it into an unwound formula, for which we can easily check whether it is satisfied or not. The game description of Tic-Tac-Toe contains the following three rules:

$$\begin{aligned} true(cell(1, 1, X)) &\rightarrow next(cell(1, 1, X)) \\ does(mark(1, 1, X)) &\rightarrow next(cell(1, 1, X)) \\ true(cell(1, 1, b)) &\rightarrow legal(mark(1, 1, X)) \end{aligned}$$

The first rule states that if the upper left cell is marked with an X then it will also be marked with an X in the next state. The second rule states that if A makes the move $mark(1, 1, X)$ then the upper left cell will be marked with an X in the next state. The third rule states that it is legal for player A to make a mark in the upper left cell if that cell is currently empty. Using these rules we obtain:
$C^\infty(N(\phi)) = true(cell(1, 1, X)) \vee (does(mark(1, 1, X)) \wedge true(cell(1, 1, b)))$
which states that if in the current state the upper left cell contains an X, or if the upper left cell is blank and player A marks it with an X, then in the next turn ϕ will be satisfied.

We now know that if the game is in a state w that contains the base-proposition $true(cell(1, 1, X))$ or it contains the base-proposition $true(cell(1, 1, b))$ and A chooses the action $mark(1, 1, X)$ then the next state $u(w, a)$ will (also) satisfy $true(cell(1, 1, X))$.

5.3 Action Normal Form

We have seen in the previous section that if we want some formula ϕ to be true in the next state, then we want $N(\phi)$ to be satisfied in the current state. However, $N(\phi)$ is non-statewise, so the satisfaction of $N(\phi)$ depends on the action chosen by the active player. Therefore, what we want is to determine, given a state w, which action a the active player needs to choose in order to satisfy $N(\phi)$. To make this easier, we transform the formula into *Action Normal Form*, as defined in [15].

Definition 17. *We say a formula ϕ is in **Action Normal Form** (ANF) for player A if it is written in the following form: $\phi = \bigvee_{t \in S} \mathcal{X}_t^\phi \wedge does(t)$ where S is a set of action-terms such that for each action $a \in \mathcal{A}$ there is a term $t \in S$ that can be unified with $\mu(a)$, and all $\mathcal{X}_t^\phi$ are statewise.*

The idea of ANF is that it explicitly separates the action-propositions from the other types of atoms. It gives us a clear recipe to determine which action to choose in a state w if we want ϕ to be satisfied.

Lemma 4. *For any non-statewise formula ϕ there is a formula $\overline{\phi}$ which is in ANF, such that:* $V, \mathcal{R} \vDash_{(w,a)} \exists\overline{\phi}$ *iff* $V, \mathcal{R} \vDash_{(w,a)} \exists\phi$.

Proof. We only prove this for the case that ϕ is ground. For each action a we can generate a formula $\mathcal{X}^{\phi}_{\mu(a)}$ by replacing every occurrence of $does(\mu(a))$ in ϕ with $\top$ and replacing all other action-propositions with $\bot$. Note that we then have that, for every $a \in \mathcal{A}$, (w, a) satisfies $\mathcal{X}^{\phi}_{\mu(a)} \wedge does(a)$ iff (w, a) satisfies ϕ.

Now suppose that for some state w and some formula ϕ we want to choose an action a such that (w, a) satisfies ϕ. We can achieve this by first generating an equivalent formula $\overline{\phi}$ which is in ANF. We can then check for every $\mathcal{X}^{\phi}_{t}$ in the expression whether it is satisfied by w or not. If for some term $t \in S$ we have that $\mathcal{X}^{\phi}_{t}$ is indeed satisfied by w then there is an action a such that $\mu(a)$ is unifiable with t and such that (w, a) satisfies $\mathcal{X}^{\phi}_{t} \wedge does(t)$ and therefore we have that (w, a) satisfies ϕ.

From now on, the notation $\overline{\phi}$ will denote any formula that is in ANF and that is equivalent to ϕ. Furthermore, we will use $\overline{\phi}^{+}$ to denote $\overline{\phi}$ in which all action-propositions have been replaced with $\top$. Thus, if: $\overline{\phi} = \bigvee_{t \in S} \mathcal{X}^{\phi}_{t} \wedge does(t)$ then: $\overline{\phi}^{+} := \bigvee_{t \in S} \mathcal{X}^{\phi}_{t}$.

Lemma 5. *If a state w satisfies $\overline{\phi}^{+}$ then there exists an action a such that (w, a) satisfies $\overline{\phi}$, and hence also ϕ.*

Proof. A state w satisfies $\overline{\phi}^{+}$ iff there is some $\mathcal{X}^{\phi}_{t}$ in the ANF of ϕ that is satisfied, which means there is a substitution θ such that w satisfies $\mathcal{X}^{\phi}_{t}[\theta]$. We then have that $(w, t[\theta])$ satisfies $\mathcal{X}^{\phi}_{t}[\theta] \wedge does(t[\theta])$, which means that (w, a) satisfies $\mathcal{X}^{\phi}_{t} \wedge does(t)$, with $a = \mu^{-1}(t[\theta])$, and therefore (w, a) satisfies $\overline{\phi}$, and because of Lemma 4 (w, a) satisfies ϕ.

Note that $\overline{\phi}$ is not uniquely defined. However, from now on we will simply assume we have some algorithm that outputs a unique $\overline{\phi}$ for any given ϕ. Then, for any odd positive integer n we can define:

$$\alpha_n = \overline{C^{\infty}(N(\alpha_{n-1}) \wedge \neg terminal)}^{+} \tag{1}$$

Lemma 6. *Let n be any odd positive integer. A state w satisfies α_n iff w is non-terminal and there is some action a for A that is legal in state w and for which the resulting state $u(w, a)$ satisfies α_{n-1}.*

Proof. If we combine Eq. (1) with Lemma 5 we conclude that w satisfies α_n iff there exists an action a such that (w, a) satisfies $C^{\infty}(N(\alpha_{n-1}) \wedge \neg terminal)$. According to Lemma 2 this holds iff (w, a) satisfies $N(\alpha_{n-1}) \wedge \neg terminal$, and then using Lemma 3 we conclude that this holds iff $u(w, a)$ satisfies α_{n-1} and w is non-terminal.

We now still need to define α_n for even n. Note that if n is even and a state w satisfies α_n then it means that in state w player B is the active player. However, since B is an adversary, the existence of an action for B that leads to a state satisfying α_{n-1} is not enough to guarantee that α_{n-1} will be satisfied in the next state. After all, B may choose a different action in order to prevent this. Therefore, for states in which B is the

active player we demand that *all* actions of B lead to a state satisfying α_{n-1}. Let us first define a formula α'_n as follows: $\alpha'_n = C^{\infty}(N(\alpha_{n-1} \vee \alpha_{n-2} \vee \ldots \alpha_0))$. Then, for n is even, we can define:

$$\alpha_n = C^{\infty}\Big(\neg terminal \wedge \neg \bigvee_{t \in S} \exists(legal(t) \wedge \neg \mathcal{X}_t^{\alpha'_n})\Big) \tag{2}$$

where S is the same set of action-terms as the one that appears in the ANF of α'_n and the $\mathcal{X}_t^{\alpha'_n}$ are also obtained from the ANF of α'_n.

Lemma 7. *Let n be an even number. A state w satisfies α_n iff for every move a of B that is legal in state w there is an integer $m < n$ such that $u(w, a)$ satisfies α_m.*

Proof. Eq.(2) states that w satisfies α_n iff w is non-terminal and there is no action-term t and no substitution θ such that w satisfies $legal(t[\theta]) \wedge \neg \mathcal{X}_t^{\alpha'_n}[\theta]$. This means that for every legal action $a = \mu^{-1}(t[\theta])$ we have that w satisfies $\mathcal{X}_t^{\alpha'_n}[\theta]$ and hence that (w, a) satisfies $\mathcal{X}_t^{\alpha'_n}[\theta] \wedge does(a)$, which means that $(w, t[\theta])$ satisfies α'_n, which is $C^{\infty}(N(\alpha_{n-1} \vee \alpha_{n-2} \vee \ldots \alpha_0))$. Again, using Lemmas 2 and 3 this means that $u(w, a)$ satisfies $\alpha_{n-1} \vee \alpha_{n-2} \vee \ldots \alpha_0$, which means there is some $m < n$ for which $u(w, a)$ satisfies α_m.

Lemma 8. *If a state w satisfies α_n and $n > 0$ then w is a non-terminal state.*

Proof. This follows directly by applying Lemma 2 to Equations 1 and 2.

Theorem 1. *If a state w satisfies α_n for some n then there exists a strategy for A that guarantees that a state satisfying α_0 can be reached in at most n steps whenever the game is in the state w.*

Proof. We know from Lemmas 6 and 7 that if w satisfies α_n and A plays optimally, then the next state will satisfy some α_m with $m < n$. This means that no matter what strategy is played by B, in every round that follows some α_m will be satisfied, and m will be decreasing with every new round. Moreover, since every α_m is non-terminal (except for $m = 0$) this means the game will eventually reach a state that satisfies α_0.

Theorem 2. *Let w be a state such that A has a strategy that guarantees that some state w' that satisfies α_0 can be reached then there is an integer n such that w satisfies α_n.*

Proof. The proof goes by induction. If w already satisfies α_0 then the theorem clearly holds. Our induction hypothesis says that for any state in which A has such a strategy in $n - 1$ steps, α_{n-1} is satisfied. We now need to prove that if w is non-terminal and A has a strategy that guarantees α_0 in n steps then α_n is satisfied. If n is odd, then A is the active player, so we know A has a move a that leads to a state $u(w, a)$ which is only $n - 1$ steps away from w'. According to the induction hypothesis we then know that $u(w, a)$ satisfies α_{n-1}. From Lemma 6 it then follows that w satisfies α_n. If n is even, then it means that all legal moves of B lead to a state that satisfies α_m for some $m < n$. According to Lemma 7 this means that w satisfies α_n.

6 Conclusions

We have presented a backward search algorithm for turn-taking games in General Game Playing. Given any desired property encoded by some formula η_A it generates a sequence of formulas $\alpha_0, \alpha_1, ...$ with the interpretation that if a game state w satisfies some α_i then player A has a strategy that guarantees that η_A will be satisfied in i steps. We expect this method to be useful as an addition to MCTS because it allows one to quickly evaluate whether a target state can be reached without having to generate all the states that lead to it. The effectiveness of this method depends on whether it is possible to keep the formulas compact. Therefore, an empirical evaluation is left as future work.

Acknowledgments

This work was sponsored by an Endeavour Research Fellowship awarded by the Australian Government Department of Education.

References

1. Alcázar, V., Borrajo, D., Fernández, S., Fuentetaja, R.: Revisiting regression in planning. In: IJCAI 2013, Beijing, China, August 3-9, 2013 (2013)
2. Bonet, B., Geffner, H.: Planning as heuristic search. Artif. Intell. 129(1-2), 5–33 (2001)
3. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering 1(1), 146–166 (1989)
4. Eiter, T., Ianni, G., Krennwallner, T.: Answer set programming: A primer. In: Reasoning Web. Semantic Technologies for Information Systems, Lecture Notes in Computer Science, vol. 5689, pp. 40–110. Springer Berlin Heidelberg (2009)
5. Finnsson, H.: Simulation-Based General Game Playing. Ph.D. thesis, School of Computer Science, Reykjavik University (2012)
6. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005)
7. Genesereth, M.R., Thielscher, M.: General Game Playing. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2014)
8. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
9. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence 6(4), 293 – 326 (1975)
10. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Proceedings of the 17th European Conference on Machine Learning. pp. 282–293. ECML'06, Springer-Verlag, Berlin, Heidelberg (2006)
11. Love, N., Genesereth, M., Hinrichs, T.: General game playing: Game description language specification. Tech. Rep. LG-2006-01, Stanford University, Stanford, CA (2006)
12. von Neumann, J.: On the theory of games of strategy. In: Tucker, A., Luce, R. (eds.) Contributions to the Theory of Games, pp. 13–42. Princeton University Press (1959)
13. Schiffel, S., Thielscher, M.: M.: Fluxplayer: A successful general game player. In: Proceedings of AAAI 2007. pp. 1191–1196. AAAI Press (2007)
14. Trutman, M., Schiffel, S.: Creating action heuristics for general game playing agents. In: The Fourth Workshop on General Intelligence in Game-Playing Agents, GIGA 2015, Held in Conjunction with IJCAI 2015, Buenos Aires, Argentina, July 26-27, 2015, Revised Selected Papers. pp. 149–164 (2015)
15. Zhang, D., Thielscher, M.: A logic for reasoning about game strategies. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI-15). pp. 1671–1677 (2015)